View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by LSBU Research Open

Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Date of publication xxxx 00, 0000, date of current version xxxx 00, 0000.

Digital Object Identifier 10.1109/ACCESS.2017.DOI

A Novel Predictive-Collaborative-Replacement (PCR) Intelligent Caching Scheme for Multi-Access Edge Computing (MECs)

EMEKA E. UGWUANYI, (Member, IEEE), MUDDESAR IQBAL, AND TASOS DAGIUKLAS, (Member, IEEE)

[1]Division of Computer Science and Informatics, London South Bank University, London SE1 0AA, (e-mail: {ugwuanye, m.iqbal, tdagiuklas}@lsbu.ac.uk)

Corresponding author: Emeka E. Ugwuanyi (e-mail: ugwuanye@lsbu.ac.uk).

This paragraph of the first footnote will contain support information, including sponsor and financial support acknowledgment. For example, "This work was supported in part by the U.S. Department of Commerce under Grant BS123456."

ABSTRACT Multi-Access Mobile Edge Computing (MEC) is proclaimed as a key technology for reducing service processing delays in 5G networks. One of the use cases in MEC is content caching as a way of bringing resources closer to the end-users. Consequently, both latency and QoE are reduced. However, MEC has a limited storage space compared to the cloud. Therefore, there is a need to effectively manage the cache storage. This paper proposes and evaluates a novel scheme (PCR) that combines proactive prediction, collaboration among MECs and replacement algorithm to manage content caching in MEC. Results show that the proposed replacement scheme outperforms conventional baseline content caching algorithms LFU, LRU, MQ, FBR, LFRU. This has been validated with experimental results using a real dataset (MovieLens20M dataset).

INDEX TERMS Edge Intelligence, intelligent caching, MEC, predictive caching,

I. INTRODUCTION

ULTRA-Reliable Low-Latency Communications (URLLC) [1] is likely the most talked-about 5G use case mainly because of the huge services it can support. URLLC aims to deliver a vastly reliable mobile wireless network with extremely low latency requirements. Low latency is tremendously important to support 5G applications [2]. This is exceptionally challenging as Cisco predicted the overall mobile data traffic is expected to grow to 77 exabytes per month by 2022 [3]. This growth is attributed to the rise of IoT and Machine to Machine (M2M) communication, use of social media and video streaming platforms, and the adoption of Augmented Reality /Virtual Reality (AR/VR) applications and 360 video streaming.

One of the ways to reduce the latency is by bringing the resources closer to the user through caching. Hence the need for Multi-Access Edge Computing (MEC) is crucial to reduce delay. MEC is defined by Taleb et al [4] as an ecosystem which aims at combining telecommunication and IT services and reducing latency by providing a cloud computing platform at the edge of the radio access network. Therefore, caching on the edge is a promising solution.

Employing an appropriate caching algorithm is pivotal to increase the overall Quality of Experience (QoE) in content distribution systems as 1% increase in hit ratio can have a positive impact [5]. The conventional caching algorithms such as First In First Out (FIFO) [6], Least Recently Used (LRU) [7], Least Frequently Used (LFU) [8], Least Frequently Recently Used (LFRU) [9] and their variants [10] [11] [12] [8] follow vastly specific rules. Therefore, these algorithms alone cannot adapt and adjust to the ever-caching user request patterns. Following the increasing popularity of machine learning and data analytics, there has been progress made on the prediction based caching algorithms [13], [14], [15], [16]. Most of these algorithms use Recurrent Neural Network (RNN) algorithms like Long Short-Term Memory (LSTM) [17] which involves data preparation and feature extraction, model training and finally cache replacement using

the trained model. The model training is usually overly time consuming and therefore is mostly done offline. However, once the model is trained with appropriate hyperparameters [18] and adequate feature engineering, it can achieve extremely high hit ratio. One major drawback is that the model created is immensely dependent on the data used on training and hence it is not hugely adaptive.

Podlipnig *et al* [19] have stated that a good caching algorithm must consider 4 factors. This includes the frequency of the cache object, the recency, the size of the cache object and the cost of retrieving the cache object. This is of no surprise as prior to this, Zhou *et al* [8] also listed 3 factors a good cache algorithm. Two among them (frequency and recency) have already been mentioned. The third factor is the temporal frequency. This is to solve the problem of cache objects that were frequently accessed in the past, obtains an excessively high frequency. As a result, such objects are never replaced even if they are no longer popular. Addressing this problem has been another motivation for this research paper.

In this paper, the aforementioned concerns of the caching algorithms have been addressed by proposing a three-fold algorithm solution to improve the cache hit ratio and access delay within a MEC environment. This includes a novel delay aware replacement caching algorithm that can find a victim in $O(1)$, a proactive online association-based caching strategy that prefetches cache objects based on anticipated user behaviour and finally a MEC collaborative caching algorithm. Experimental results show that the proposed scheme outperforms conventional algorithms with regards to hit ratio. Additionally, it outperforms an offline caching algorithm with a pre-trained model.

The remainder of the paper is structured as follows. The review of related relevant work is presented in section II. Section III explains the system architecture and system model. Section IV details the proposed algorithms. Section V presents the experimental results and evaluates the performance of the algorithms. Finally, the paper concludes in Section VI and possible future works have been included in section VII.

II. LITERATURE REVIEW

Caching algorithms have been studied widely by the research community in different research fields including MEC. This ranges from operating systems to network caching and in-between. In this section, a review is carried out on the caching algorithms that are relevant to this study.

A. CONVENTIONAL REPLACEMENT ALGORITHMS

In this section, the popular conventional algorithms that are still commonly used have been reviewed. The vast aim of these algorithms is to increase the hit ratio.

1) First in First Out (FIFO)

This is one of the simplest replacement policies in terms of time complexity and implementation. In a FIFO queue, cache objects are placed in the tail of the queue. If there is a need for replacement, cache objects are removed from the head until there is enough space for the incoming request. The time complexity of this algorithm is $O(1)$.

2) Least Recently Used (LRU)

LRU [7] is still one of the commonly used algorithms. When the cache is full, the policy replaces the cache object which has not been referenced for the longest of time. The strategy is based on the observation that blocks that have been recently referenced are likely to be used again in the future. LRU works well with workloads that exhibit strong temporal locality. However, according to [20], LRU does not work well with file server caches. The time complexity of LRU algorithm is $O(1)$.

3) Least Frequently Used (LFU)

LFU is another classic cache replacement algorithm. LFU maintains a reference count for all cache objects. Therefore, when the cache is full it replaces the cache object with the lowest reference count. The rationale of the algorithm is that some cache blocks are more frequently accessed than others. Therefore, the frequency count is a good estimate of the probability of a cache object to be requested. LFU has two main drawbacks. Firstly, there may be a tie if two cache objects have the same frequency. Secondly, a cache object may accumulate large reference count and never replaced even if the cache object is no longer active. There have been many improvements proposed to address the drawback of LFU. One of these improved versions is the aged LFU. This policy gives different weight to recent and old references. Aged LFU performs better than the original LFU [8]. The time complexity of LFU is $O(\log(n))$.

4) Least Recently Used K(LRU-K)

This algorithm combines both LFU and LRU schemes. It was first introduced in database disk buffering. The basic idea of LRU-k [10] is to keep track of the times of the last K references to popular cache objects. This information is then used to statistically estimate the interarrival times of references on a request by request basis. The replacement decision is based on the reference density observed during the past K references. When K is small, cold cache objects are identified quicker as such cache objects have a wider span between the current time and the k^{th}-to-last reference time. The time complexity of LRU-K is $O(\log(n))$.

5) Least Recently Frequently Used (LRFU)

LFRU [9] is another algorithm that combines LFU and LRU. The strategy of this algorithm is to replace cache objects that are least frequently used and not recently used. Each cached object is associated with a Combined Recency and Frequency value (CRF). The cache object with the lowest CRF value is replaced. Each request of a cache object contributes to its CRF. LRFU has a time complexity of between $O(1)$ and $O(\log(n))$.

6) Frequency Based Replacement (FBR)

FBR [11] algorithm is a hybrid replacement scheme which combines both LRU and LFU to capture the benefits of both algorithms. FBR has been initially proposed for managing caches for file systems, management systems or disk control units. FBR maintains LRU queues of cache objects with the same frequency count. To address the problem of cache objects accumulating large reference counts, the algorithm maintains 3 sections. These include F_{new}, F_{middle} and F_{old}. These sections are used to bound the frequency count of certain cache objects and they are required algorithm parameters. FBR also requires 2 additional parameters, A_{max} and C_{max}. A_{max} refers to the maximum average of frequency counts to be maintained while C_{max} is the maximum chain count. More details can be found in [11]. The replacement decision is primarily based on the frequency count. According to [20], FBR is the best algorithm compared to LRU and LFU. The time complexity of FBR ranges from $O(1)$ to $O(\log(n))$.

7) Two Queue (2Q)

2Q algorithm [12] has been proposed as an improvement to LRU-k. The motivation is to reduce the access overhead and remove cold cache objects quickly. The 2Q uses two LRU queues $A1_{out}$ and A_m and an additional FIFO queue $A1_{in}$. The cache objects are initially stored in the $A1_{in}$ when first accessed. When the cache is evicted from $A1_{in}$ it is then added to $A1_{out}$. If a cache object in $A1_{out}$ is accessed, it is moved to A_m. The authors have proposed a scheme to select the efficient sizes for both $A1_{in}$ and $A1_{out}$. The 2Q performs better than FBR, LRU and LFU for second-level buffer caches [8]. The time complexity of 2Q is $O(1)$.

8) Multi-Queue (MQ)

MQ [8] has a comparable technique to 2Q. The motivation is to create an algorithm that supports minimal lifetime for cache objects, has a frequency-based priority and supports temporal frequency. The MQ uses m number of LRU queues, where m is a parameter. Cache objects in certain queues have a longer lifetime than others depending on the queue the object lies. MQ also uses a history FIFO queue Q_{out} of limited size to store recently evicted cache objects. MQ evicts the cache object in the tail of the LRU queue with the least frequency. MQ [8] performs better than FBR, Q2, LRU and LFU. The time complexity is $O(1)$.

B. NETWORK AWARE REPLACEMENT ALGORITHMS

Many factors affect the performance of the replacement algorithms used in network caching. These include the requested object size, latency, bandwidth, miss penalty, temporal locality and long-term access frequency. Successful application of these algorithms can reduce network traffic, response time, and server load. The algorithms reviewed in this section take at least one of these parameters in consideration during cache replacement.

1) GreedyDual (GD) Algorithms

GD Algorithm has several variations, but the key objective is to replace the cache object with the lowest cost value based on a specified cost function. These variations have different cost functions. The original GreedyDual [21] has been proposed by Young. The motivation of the algorithm has been to deal with cache objects that have the same size but incur a different cost in bringing it to the cache-store. When a cache object is retrieved, a value H is assigned to it. This is the cost of bringing the cache object to the cache-store. The algorithm replaces the cache object with the min H and then all cache objects reduce their H value by min H. The time complexity for this is $O((n-1) \times \log(n))$. Two other variations of GD are listed below:

a: GD-Size

GD-size [22] extends the original GD by adding the size of the cache object to the cost function. Therefore, $H = cost/size$, where $size$ refers to the cache size and $cost$ could vary depending on the cache priority. The $cost$ could be set to 1 if the goal is to maximize the hit ratio. It could be set to the downloading latency if the goal is to minimize average latency. Finally, it could be set to the network cost if the goal is to minimize the total cost. The time complexity of GD-Size $O(\log(n))$.

b: GreedyDual-Size with Frequency (GDSF)

GDSF [23] has been proposed as an extension of GD-Size. The limitation of GD-Size is that it does not consider the popularity of the cache objects during cache replacement. GDSF has $H = F * (cost/size) + L$, where F is the frequency count and L is a running age factor. L starts at 0 and is updated for each replaced object if the priority key of this object in the priority queue.

2) Least Unified-Value (LUV)

LUV [24] allocates a calculated value to each cached object. When the cache is full, the cache object with the lowest value is replaced. The value is calculated by $weight \times H$, where $weight$ is the retrieval cost $(cost/size)$ and H is the probability that the object is going to be re-referenced in the future. The time complexity of LUV is $O(\log(n))$.

3) Lowest-Latency-First (LLF)

LLF ranks the cache objects based on its download latency. When the cache is full it replaces the cache object with the lowest latency. The motivation of this scheme is to minimize the total latency in the system. The time complexity of the algorithm is $O(\log(n))$.

C. SIZE AWARE ALGORITHMS

In this section, algorithms that predominately makes cache replacement decisions based on the requested object size are reviewed.

1) Size

The Size algorithm [23] replaces the largest cache object when the cache is full. The strategy is to increase the cache hits by increasing the number of cache objects in the cache queue. Therefore, to minimize the miss ratio, one large object is replaced rather than many smaller ones. The limitation of this approach is that the smallest cache objects which are rarely accessed are never replaced.

2) Size-Adjusted LRU

The size-adjusted LRU [23] associates a cost-to-size ratio to each of the cached objects. The cost value is a function of the size and access time of the cache object. The cost-to-size ratio is $\frac{1}{Size \times \Delta T}$. ΔT is the elapsed time from last access time to current time. The time complexity of the algorithm is $O(\log(n))$.

3) Least Recently Used – Size adjusted and Popularity aware (LRU-SP)

LRU-SP [25] uses two extensions of the LRU algorithm, namely Size-adjusted LRU and segmented LRU. The cost to size function of LRU-SP is $(nref/(size \times \Delta T))$. Cache objects are put into a limited number of groups according to $[\log(size/nref)]$. When the cache is full, only the last 20 cache objects is considered for replacement. Therefore, the cache object with the lowest cache-to-ratio value is replaced. The time complexity of the algorithm is $O(1)$.

4) Log(Size)+LRU

Log(Size)+LRU [23] uses $\log(size)$ as a cost function. Therefore, it evicts the cache object with the largest $\log(size)$ and is the least recently used.

5) Pitkow/Recker

Pitkow/Recker [23] removes the least recently used cache object, except if all cache objects are accessed within a given time interval, in which case the largest one is removed.

D. EDGE CACHING ALGORITHMS

There has been a considerable amount of research carried out on MEC to enhance its cache performance. To accomplish this, the caching algorithm must be designed to support cache sharing among MEC nodes, reduce latency and bandwidth and increase network robustness and reliability. In this section, such relevant algorithms have been reviewed.

Wu *et al* [26] have proposed a collaborative edge caching mechanism for ICN. The scheme advocates cache redundancy by replicating cache object to the next hop whenever a cache hit occurs. Ndikumana *et al* [27] have proposed a collaborative scheme for edge computing with a collaborative space defined by the network administrator based on hop count distance between edge nodes. In the proposed scheme, the edge nodes periodically exchange resource and cache updates. Chen *et al* [28] have proposed a content popularity prediction on the edge based on neural collaborative filtering. In [29], an inter and intra tier collaborative hierarchical caching mechanism over 5G edge computing have been proposed. The scheme makes caching decisions to minimise the number of wireless hops in obtaining a cache object while maximizing the hit ratio. Liu *et al* [30] have proposed a collaborative online edge caching algorithm. The scheme uses a Bayesian clustering technique to group users based on their request preference. Popular preferences are then cached to improve the global cache hits. Saputra *et al* [31] have proposed two proactive and cooperative caching framework for mobile edge network. In the first approach, the edge nodes send data to a central server, which then creates a deep learning model based on the data popularity and sends it to the edge servers. The second approach allows each edge node to create a local model and then send it to the central server for model aggregation. The aggregated global model is then sent back to the edge nodes.

Chen *et al* [32], have proposed a neural collaborative filtering caching strategy for edge computing. The proposed method incorporates a greedy algorithm, popularity prediction algorithm and a content cache replacement algorithm. Simulation results show that the proposed algorithm can outperform baseline algorithms with regards to hit rate, transmission delay and content cache space utilization

Xu *et al* [33] have proposed a hybrid edge caching scheme for tactile internet in 5G. The proposed scheme has been aimed at energy efficiency improvement in proactive in-network caching. The cache replacement policy proposed assumes that the cache files follows Zipf distribution. Simulation results have shown that the proposed method achieves better latency compared to conventional caching algorithms.

E. PREDICTIVE CACHING ALGORITHMS

The optimal caching algorithm is an algorithm that can accurately predict the cache request pattern for $t+1$, where t is the current time and use this to make appropriate caching decisions. For such algorithms, the prediction cost is usually expensive, and many at-times fail to be consistently accurate. Therefore, there are only a handful of such caching algorithms. These algorithms are reviewed in the following paragraphs.

Qi *et al* [34] have proposed a proactive caching scheme for the wireless edge. The proposed scheme applies federated learning for cache popularity prediction. The authors highlighted a security vulnerability with centralised learning as it needs to collect information from users during the learning process which can be personal. Utilizing the proposed approach, each user uploads a weighted sum of preference and file popularity to the base station where models are aggregated. Simulation results show that the proposed scheme achieves a close cache-hit ratio to a centralised learning approach.

Tan *et al* [35] have proposed a reinforcement learning-based optimal computing and caching scheme for edge networks. The problem has been formulated as an infinite-horizon average-cost Markov decision process (MDP). The

authors aimed at maximizing the bandwidth utilization and decreasing the quantity of data transmitted. The scheme considers long-term file popularity and short-term temporal correlations of user requests to fully utilize bandwidth. Simulation results show that the proposed policy scheme can predict content popularity and user future demands.

Dutta *et al* [14] have proposed a caching framework for mobile wireless networks. The proposed technique uses a predictive scheme for both replacement and cache prefetch. The problem has been formulated as a QoE optimization problem and solved using Markov Predictive Control and Markov Decision Process. Prediction is done using the FP-Growth association rule-based algorithm. Empirical results have shown that the proposed algorithm performs better than LFU, LRU and FIFO in terms of hit ratio.

Chan et al [15] have proposed a big data-driven predictive caching at the wireless edge. The framework have applied a machine learning-based approach to anticipate user behaviours and content patterns and then prefetch content with expected high popularity. The framework uses a generic Markov prediction model for prediction. The authors state that the machine learning-based approach is useful in improving the cache performance when the popularity distribution fails to follow Zipf distribution. The proposed algorithm performs better than the LRU algorithm.

Rahman *et al* [36] have proposed a deep learning predictive caching framework for edge networks. The proposed framework uses a Long Short-Term Memory (LSTM) Recurrent Neural Network (RNN) model for predicting the popularity of cache objects. The authors have used MovieLens 20M [37] dataset for training the model. The authors assume there will be little changes in the dataset and therefore they have not considered model update during runtime. The model error rate has been evaluated. However, the proposed algorithm hit rate ratio has not been analysed.

To address the problem of predictive model inaccuracy due to changing popularity distribution of cache objects, Song et al [38] have proposed a dynamic content placement caching framework. The novel learning framework predicts the temporal cache distribution of future cache contents. The content placement is periodically updated based on future requests. Empirical results have shown that the algorithm performs better than conventional online caching algorithms.

F. CONTRIBUTIONS

The contributions of this article can be summarized as follows:

- A novel cache replacement strategy has been designed which selects victim based on popularity, recency and network cost. The scheme can identify *cold* cache objects through its selective caching approach. Additionally, the problem of temporary frequency has been addressed.
- A proactive caching algorithm has been designed that utilises association rule mining to predict cache behavioural patterns based on historic events. Furthermore, cache objects are prefetched when necessary to optimise cache storage and reduce network delay.
- A collaborative caching strategy has been designed for sharing and retrieving cache among MECs while reducing cache redundancy in the cooperative space.

III. SYSTEM MODEL

MEC aims to reduce the load at the core network by bringing computational and caching resources closer to the users. Mobile operators and content providers would benefit from caching popular content in the MEC local cache to improve the QoE of the users. Let's consider a typical system architecture as shown in Figure 1. Here, there is a set of MECs in a cluster that defines a collaborative space to support the core network. Let $C = \{C_1, C_2, C_3..C_n\}$ denote a set of collaborative spaces. Each collaborative space contains a set of MEC servers $C_i = \{M_1, M_2, ..M_n\}$. Without loss of generality, it has been assumed that the MEC is co-located with the base station to provide computational and caching resources. It is assumed that the collaborative space is defined by the network administrator based on the hop count distance between the MEC nodes [27]. This is done to reduce the communication delay within the collaborative space and reduce the communication overhead. The users are connected to the MEC in the collaborative space closet to them. The MEC maintains a disjoint one-to-many cardinality with the UE. Here an edge node is connected to many UE, but no UE is connected to multiple edge nodes. Lets denote $U_i = \{u_1, u_2, ..u_n\}$ as a finite non-empty set of UEs connected to an MEC M_i. Each u_i request data r_i to be retrieved through the MEC where $r_i \in R$. $R = \{r_1, r_2.., r_n\}$ is a finite non-empty set of data that can be retrieved from C or P, where P denotes the cloud platform. Requests that cannot be retrieved from C are sent to P through the core network.

IV. PROPOSED SCHEME (PCR)

In this section, the problem of temporal frequency has been addressed. Additionally, a proactive predictive caching scheme is proposed that learns user's request pattern, anticipates requests and prefetches it, is detailed. Finally, a collaborative algorithm is proposed for effective utilization of the global MEC cache storage. In the following sections, the details of the proposed schemes are outlined.

The proposed caching framework is depicted in Figure 2. Following the numbered items in the diagram, the algorithm is initialised with a new user request. If the requested cache object is not in the local cache, then the MEC *Collaborative Scheme* is used to retrieve the object. If it is not in the collaborative cache, then it is retrieved from the content server. The *Replacement Scheme* handles the identification and eviction of the cache victim when the cache is full. Finally, the *Cache Prediction Scheme* generates cache sequential association rules based on the received request patterns. Therefore, when there is a match based on the rules generated, the cache ob-

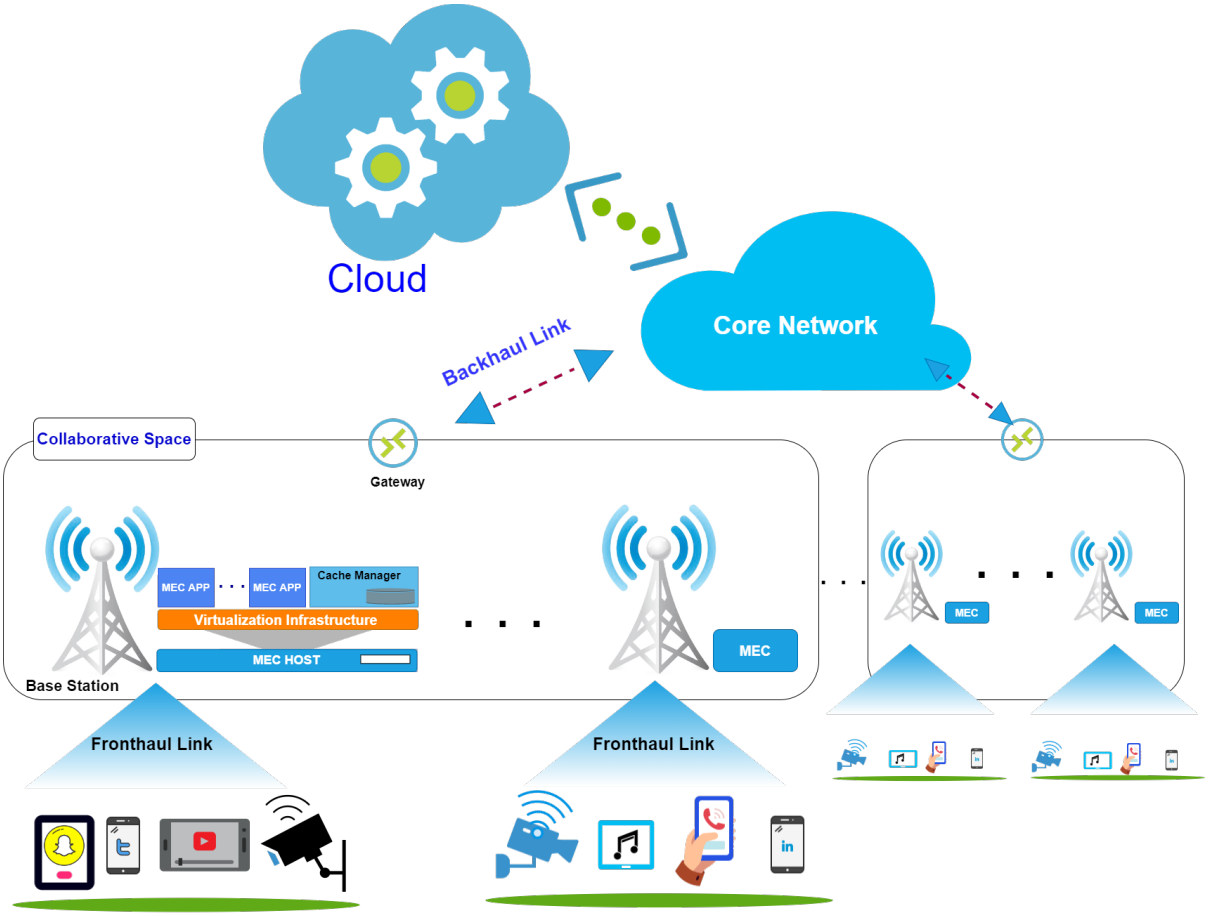

FIGURE 1. System Architecture for MEC Collaborative Content Caching

jects are prefetched and stored in the local cache. These three schemes are explained in detail in the following sections.

A. REPLACEMENT SCHEME: SELECTIVE HISTORIC LEAST FREQUENTLY USED (SHLFRU)

The rationale of the proposed content caching algorithm is to design and develop an efficient caching algorithm with competitive time complexity. The basis of SHLFRU is in the combination of LRU and LFU algorithm, in which the least frequent and least recent cache object is replaced. However, two modifications have been made to this algorithm.

1) Selective caching

The problem of *cold* cache objects, which are requested once and not requested again for an overly long time and therefore not useful of being the cache-store is addressed. To solve this, a selective caching approach is used where the requested cache object r_i is only cached on either two conditions. The first condition is based on temporal locality, thus cache objects with a higher temporal locality have higher priority. This has been achieved using a history queue. The second condition prioritises cache objects with higher retrieval cost. This is to reduce user access cost. The two conditions are

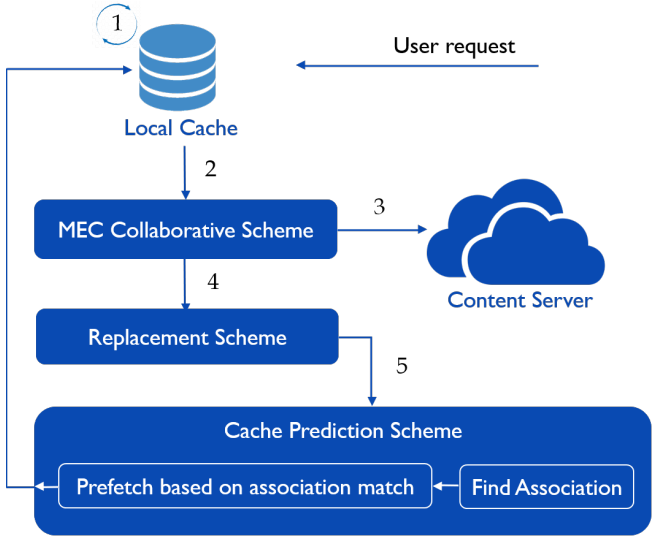

FIGURE 2. PCR Caching Framework

represented as in equations 1 and 2.

$$r_i^{(R-1)} \leq r_j^{(R-1)} \; \forall \, r_i \in H \tag{1}$$

$$r_i^c > r_j^c \tag{2}$$

Here, H is the history queue, $r_i^{(R-1)}$ and $r_j^{(R-1)}$ are the previous recency of the new cache object r_i and the least frequently and recently used object respectively. r_i^c is the cost of retrieval of r_i and r_j^c is the cost of retrieval of the r_j.

$$r_i^c = \frac{size}{t_{rate}} \tag{3}$$

Here, $size$ is the size of r_i in bits and t_{rate} is the transmission rate. r_i^c is taken to be the miss penalty, which is the delay in retrieving the cache object if there is a cache miss. The history queue H is a FIFO queue of a finite size h_{size}. It keeps the details of recently evicted blocks but not the cache data. The selective caching decision is only activated when the MEC cache store M_i^{store} is full.

$$|M_i^{store}| \geq cache_{size} \tag{4}$$

if either equation 1 or 2 is true, then r_i is removed from H and is pushed into M_i^{store}.

2) Temporal Frequency

To address the problem of temporal frequency, a frequency count bounding strategy is used. In this approach, the maximum distance between two consecutive cache objects r_j and r_i in an LRU queue is grouped by their frequency count that is bounded by f_{max}. With this approach and r_i being the lead cache object in M_i^{store}, the increment of the frequency count of a cache object r_i^f is determined by the function $FreqCount(r_i^f)$ stated in the equation below.

$$FreqCount(r_i^f) = \begin{cases} r_i^f + 1, & r_i^f - r_j^f < f_{max} \\ r_i^f, & r_i^f - r_j^f \geq f_{max} \end{cases} \tag{5}$$

Additionally, to avoid the problem of overflow associated with the practical implementation of frequency counts, a similar technique used in FBR [11] is applied. In this approach, the sum of all the frequency count f_{sum} is dynamically maintained. Therefore, every frequency count in M_i^{store} is reduced whenever the following condition occurs.

$$\frac{f_{sum}}{|M_i^{store}|} \geq A_{max} \tag{6}$$

A_{max} is a predefined maximum value which is a parameter of the algorithm. Here, small A_{max} means high frequency updates. The frequency count of the cache objects in H and M_i^{store} is reduced using the following equation according to [11].

$$\left\lceil \frac{r_i^f}{2} \right\rceil \forall\, r_i \in \{M_i^{store}, H\} \tag{7}$$

Using this approach, in a steady-state, f_{sum} would lie between $|M_i^{store}| \times \frac{A_{max}}{2}$ and $|M_i^{store}| \times A_{max}$. Note that in this reduction a count of one will remain at one, a count of two would be two, a count of three would be 2, etc.

TABLE I. TIME COMPLEXITY COMPARISON

Algorithm	Time complexity
SHLFRU	$O(1)$
MQ	$O(1)$
LRU	$O(1)$
LFU	$O(\log(n))$
LFRU	$O(\log(n))$
FBR	$O(\log(n))$

The SHLFRU algorithm uses multiple LRU queues Q to achieve LRFU where each LRU queue Q_i contains cache objects with the same frequency count.

$$Q = \{Q_1, Q_2.., Q_n\}\ s.t\ \forall\ r_i \in Q_i, r_i^f = i \tag{8}$$

The reference to the LRU queue that contains the cache objects with the least frequency Q_L is dynamically maintained. Therefore, when a cache needs to be replaced, the victim is the cache object in the tail of Q_L.

3) Time Complexity

Maintaining an LRU queue requires a tail insertion/head taking and incurs no overhead. Since Q_L is maintained, a heap data structure is not required to keep the LFU stored. Hence, the replacement victim can be found in constant time. If Q_L changes a maximum of f_{max} queries are required to find the next Q_L. f_{max} is a constant that does not depend on the scales. In all, the time complexity of SHLFRU is $O(1)$. The time complexity comparison is depicted in TABLE I Taking these updates into consideration, the proposed algorithm SHLFRU is depicted in Algorithm I.

4) SIMULATION EXPERIMENTS

To evaluate the efficiency of the proposed content caching algorithm, SHLFRU has been compared with existing algorithms. These algorithms have been implemented using *Python*. The simulation implementation project is available on *GitHub* [39] and the online application is available [40]. The purpose of the simulation is to evaluate the hit ratio of SHLFRU compared to existing algorithms. These algorithms are evaluated with varying Zipf-popularity distribution parameter (α) and $cache_{size}$. SHLFRU, FBR and MQ require additional parameters. The experimental parameters are summarized in TABLE II. The parameters of FBR [11] and MQ [8] have been explained in section II. Seven algorithms have been evaluated including LFU, LRU, FIFO, MQ, FBR and OPT.

5) Results

The results obtained from the simulation is displayed in Figure 3. It can be deduced that SHLFRU performs better than the compared algorithms. Its performance is robust for different workloads and cache sizes. It can also be seen that MQ performs better than other algorithms except SHLRFU.

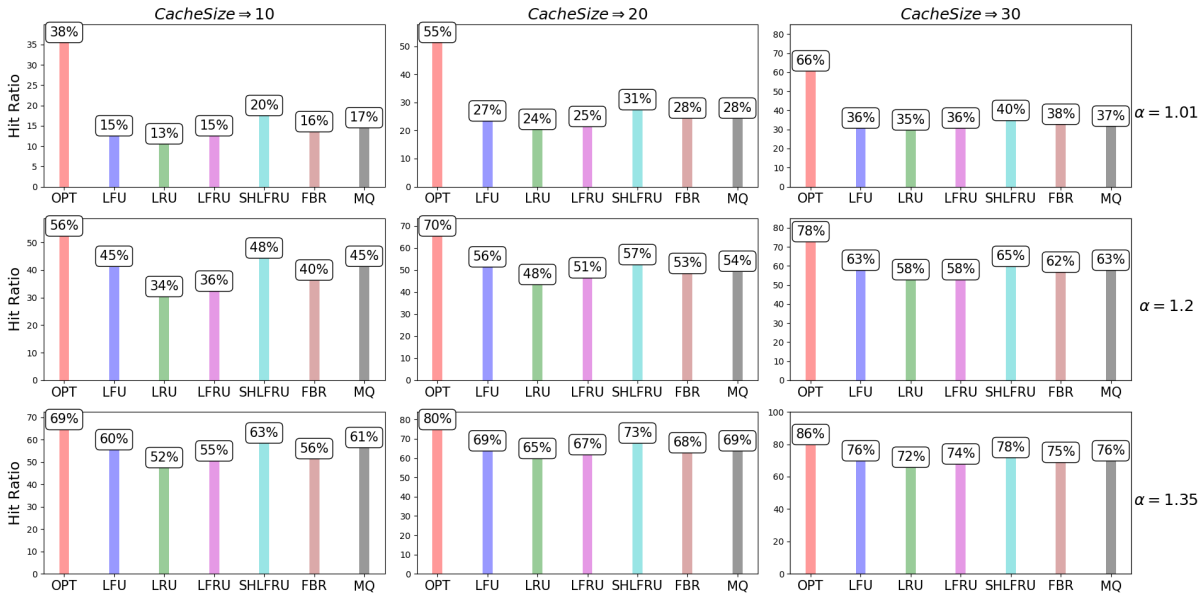

FIGURE 3. Simulation Result of the comparison of Algorithms with varying Zipf parameter (α) and Cache size

TABLE II. EXPERIMENTAL PARAMETERS

Algorithm	Parameter	Value
SHLFRU	A_{max}	100
	f_{max}	10
	h_{size}	$4 \times cache_{size}$
FBR	F_{new}	30%
	F_{old}	30%
	A_{max}	100
	C_{max}	11
MQ	$\|Q_{out}\|$	$4 \times cache_{size}$
	m (No of LRU Q)	8
General	α	{1.01,1.20,1.35}
	$cache_{size}$	{10,20,30}
	No of requests	5000
	No of content	100

SHLRFU maintains at least a 3% (3% of 5000 requests is 1500) improvement in hit ratio compared to MQ across the experiment. This improved performance can be attributed to the selective caching of SHLRFU and its ability to quickly identify *cold* cache objects. FBR performs almost as good as MQ across the simulation results. LFU surprisingly performs better than LRU across the simulation results.

B. CACHE PREDICTION SCHEME: PROACTIVE PREFETCH CACHING ALGORITHM (PPCA)

The cache store would be efficiently managed if the users' future request pattern is known. This is proved from the simulation results obtained in Figure 3, as OPT performs way better than other algorithms. The best way to predict the future is to study the past.The motivation of this proactive caching algorithm is to propose a novel predictive caching strategy based on learning the association patterns between content request in a MEC environment, where the content popularity is time-varying and unknown. Association rule mining techniques are leveraged to identify content requests with close relations. Here, for a given MEC M_i and time t_n, request history Req, from time t_{n-k} to t_n is utilized to generate rules $(Rules)$ that maps antecedents $rule^{ant}$ to consequents $rule^{cons}$. This approach is employed rather than making predictions for t_{n+1}. It is assumed that certain content requests con_i and con_j are often requested together or sequential by users, where con_i and con_j are sets of variable lengths. Therefore, the aim is to classify con_i and con_j as either $rule^{ant}$ or $rule^{cons}$ such that

$$con_i \rightarrow con_(j) \neq con_j \rightarrow con_i \tag{9}$$

Let S be the FIFO request sequence with length m which has been received by a MEC M_i.

$$S = \{r_1, r_2..r_m\} \tag{10}$$

This problem is classified as an association sequential pattern mining. Here the order of requests are maintained but no duplicate items may appear in the sequence. To reduce the processing time of the algorithm, two pruning techniques are employed. First, the dimension of the request sequence to be mined s_i is bounded by k.

$$s_i = \{r_1, r_2...r_k\}\ s.t\ s_i \subseteq S \tag{11}$$

k is dynamically obtained from the number of unique elements u_{no} for a given window size w_{size}, such that the sequence bounded by the window size s_{ws} is a subset of S. Therefore, k is deduced from the following equation.

$$k = (u_{no})^2 \tag{12}$$

This is to have enough training data that can generate meaningful insights. Similar approach has been used by [15].

Algorithm I SHLRFU

Input $cache_{size}, A_{max}, f_{max}, h_{size}$
Output $None$
Initialization
$M_i^{store} \leftarrow \varnothing$
$H \leftarrow \varnothing$
$cache_{decision} \leftarrow True$
// Procedure to be invoked upon reference to cache object r_i

if $r_i \in M_i^s tore$ **then**
  $Q_i.pop(r_i)$
else
  $D_1 \leftarrow do\ eq.(1)$
  $D_2 \leftarrow do\ eq.(2)$
  if $D_1\ or\ D_2\ is\ True$ **then**
    $Victim \leftarrow Q_L.pop()$
    $H.push(Victim)$
    if $r_i\ not\ in\ H$ **then**
      $H.push(r_i)$
      $cache_{decision} \leftarrow False$
    else
      $update\ r_i^{recency}$
      $cache_{decision} \leftarrow False$
    end if
  end if
end if
$k \leftarrow FreqCount(r_i^f)$
if $cache_{decision}$ **then**
  if $Q_k\ not\ in\ M_i^s tore$ **then**
    $Q_k \leftarrow \varnothing$
  end if
  $Q_k.push(r_i)$
end if
if $eq(6)\ is\ True$ **then**
  $do\ eq(7)$
end if

In this regard, the training dataset D is a matrix obtained from S with a dimension of $u_{no} \times u_{no}$. Therefore, for the association mining to be performed, the following condition must be met.

$$k \leq m \tag{13}$$

Secondly, the minimum support threshold $support_{min}$ is used to reduce the number of item-sets to be evaluated as candidates during the association mining. The support of a given sequence set $con_i\ s.t\ con_i \subset D$ is the number of rows n_{row} in D that contain con_i.

$$support(con_i) = \frac{n_{row}}{|D|} \tag{14}$$

Therefore, given $support_{min}$ a set S_{con} that contains all con_i that satisfies $support(con_i) \geq support_{min}$ is sort after.

$$S_{con} = \{con_1, con_2..con_n\} \forall support(con_i) \geq support_{min} \tag{15}$$

Using S_{con}, $Rules_{all}$ is generated that contains rules with antecedents which are a subset of S_{con}.

$$Rules_{all} = \{rule_1, rule_2..rule_n\}\ \forall\ rule_i^{ant} \subseteq S_{con} \tag{16}$$

The generated rules $Rule_{all}$ are then ranked to evaluate the strongest rules using the rule support $(Rule_{support})$ and rule confidence $(Rule_{confidence})$ [41]. $Rule_{confidence}$ of $(con_i \rightarrow con_j)$ is the proportion of transactions in D including both con_i and con_j.

$$Rule_{confidence}(con_i \rightarrow con_j) = \frac{Support(con_i \cup con_j)}{Support(con_i)} \tag{17}$$

Given the sorted ranked rules $Rule_{ranked}$, the top p rules are selected to be used for prefetch caching.

$$Rule_{output} = \{rule_1, rule_2...rule_p\}\ \forall\ rule_i \in Rule_{ranked} \tag{18}$$

Algorithm II PPCA

Input: $S, w_{size}, m, support_{min}, mem_{max}, Rule_{outpu}t^{max}, p$
Output: $Rule_{output}$
Initialization:
$Rule_{output} \leftarrow \varnothing$

From $s_w s\ obtain\ u_{no}$
$k \leftarrow (u_{no})^2$
if $k \leq m$ **then**
  $s_i = \{r_1, r_2...r_k\}\ s.t\ s_i \subseteq S$
  obtain D from s_i *s.t* $dimension = u_n o \times u_n o$
  using eq.(15) obtain $S_c on$
  if eq.(19) is *false* **then**
    $Rule_{all} \leftarrow FPGrowth(D)$
  else
    $Rule_{all} \leftarrow Apriori(D)$
  end if
  $Rule_{ranked} \leftarrow sort(Rule_{all})$
  $Rule_{output} \leftarrow slice(Rule_{ranked}, p)$
end if

$Rule_{output}$ is updated every $Rule_{timeout}$ to ensure that the most relevant rules are stored. After the completion of each user request, the $rule^{cons}$ of the $rule^{ant}$ that matches $Req((t-l) \rightarrow t)$ is prefetched. If there is no match, no prefetch is done. Here, t is the current position in Req and l is the number of elements in $rule^{ant}$. To ensure quick lookup, the rules in $Rule_{output}$ is stored in a hash table with the key being the number of elements in $rule^{ant}$. The value is also a hash table with the key being the $rule^{ant}$ and the value being the $rule^{cons}$. The number of rules in the $Rule_{output}$ is

bounded by $Rule_{output}^{max}$. This is the maximum number of elements in the $rule^{ant}$ that can be stored in $Rule_{output}$. Thus, the maximum look-up done to find matches is $Rule_{output}^{max}$.

Two major algorithms can be used for association rule mining which are Apriori [42] and FP-Growth [43]. Apriori generates better association with sparse dataset but it is memory intensive due to its breadth-first approach. FP-Growth is quicker and uses a depth-first approach. However, FP-Growth does not perform well with sparse datasets [44]. The best of both worlds has been combined by employing both algorithms and then choosing which one to use at runtime based on the sparse density and a given memory threshold mem_{max}. In this approach, the default algorithm is FP-Growth. However, the average memory utilized mem_{avg} when updating $Rule_{output}$ is stored. Therefore, Apriori is used if the following equation is satisfied, where M_i^{mem} is the current memory utilization of the MEC and $D^{density}$ is the sparse density of the dataset D.

$$Apriori \mid M_i^{mem} < mem_{max} \; \forall \; D^{density} > 0.5 \qquad (19)$$

1) Time Complexity

The time complexity of the association mining is $O(2^n)$, where n is the number of elements in D. The time complexity of ranking the algorithm is $O(\log(n))$. The total runtime is $(2^n + \log(n))$. Therefore, the time complexity is $O(2^n)$.

C. MEC COLLABORATIVE SCHEME: COLLABORATIVE GREEDY ALGORITHM

The motivation behind this scheme is to efficiently manage the cache in the collaborative space by reducing data redundancy and increasing the sharing of cache data among MECs. The aim here is to increase the efficiency of the global collaborative cache by improving the efficiency of the individual edge node. Let's assume content-centric networking for sharing cache data within the collaborative space. Therefore, contents are retrieved using a named identifier n_i. Additionally, let's assume a name resolution server nrs is located in each collaborative space C_i. The contents in nrs are populated by the MECs in C_i after each content retrieval. To ensure security and integrity, the contents in the nrs is stored in a blockchain. M_i stores $M_i^{names} | M_i^{names} \subset nrs$ locally to improve efficiency. M_i^{names} is a FIFO queue with limited size. For simplicity, let's assume each MEC M_i has homogeneous storage capacity and belongs to a collaborative space C_i. The total content stored in the collaborative store is denoted by C_i^{store}. Thus, $M_i^{store} \subset C_i^{store}$. The C_i^{store} is populated by event-driven updates sent by MECs. Furthermore, a binary cache function $Cf(cache_{store}, r_i)$ is defined that indicates if a cache object is available in a cache-store.

$$Cf(cache_{store}, n_i) \in \{0, 1\} \qquad (20)$$

From eq.(20), 1 implies that n_i is cached in a given $cache_{store}$ and 0 otherwise. If a named content is not stored in either M_i^{store} nor C_i^{store}, it is retrieved from the content provider in the cloud P. However, if n_i is in more than one MEC in C_i then n_i is retrieved from the MEC with the least network delay. Let's denote the network delay between two MECs M_i and M_j as $M_{i\to j}^{delay}$. If $M_i^{n_i}$ denotes a set of MECs that have n_i in the cache and M_d is the MEC with the least network delay. The collaborative cache retrieve function $Retrive(n_i, M_d)$ has been defined in eq.(21).

$$Retrive(n_i, M_d) \leftarrow \; \forall \; n_i \in M_i^{n_i} \; s.t \; |M_i^{n_i}| > 1 \qquad (21)$$

To reduce cache redundancy, the number of MECs that can store n_i is capped at w percent. Therefore, the total number of MECs TNM that can store n_i is represented in eq.(22)

$$TNM = w \times |C_i| \qquad (22)$$

Algorithm III Collaborative Offloading Decision Algorithm

Input w
Output $None$
Initialization
$M_i^{names} \leftarrow \varnothing$
$M_i^{store} \leftarrow \varnothing$
$C_i^{store} \leftarrow \varnothing$
$TNM \leftarrow w \times |C_i|$
// Procedure to be invoked upon reference to cache object r_i

if $r_i \in M_i^{store}$ *not* $True$ **then**
  $n_i \leftarrow M_i^{names}(r_i)$
  $q = Cf(C_i^{store}, n_i)$
  if $q \in \{1\}$ **then**
    if $|M_i^{n_i}| > 1$ **then**
      $r_i \leftarrow Retrive(n_i, M_d)$
    else
      $r_i \leftarrow Retrive(n_i, M_j)$
    end if
    if $|M_i^{n_i}| < TNM$ **then**
      $M_i^{store}.push(r_i)$
    end if
  end if
else
  use Algorithm I to fetch r_i
  Send push request to nrs with r_i
  send cache update to C_i
end if
Use Algorithm II to prefetch cache

V. EXPERIMENTATION

To evaluate the efficiency of the proposed algorithm PCR, an emulation environment have been implemented which consists of a varying number of MECs $\{6, 8, 10\}$ and a content server. The content server has been deployed on *Netlify* [45] and the MECs have been deployed on GNS3 platform. Each MEC is a Linux Server utilizing docker as its Virtualization infrastructure. Communication among the MECs is done using a messaging broker. The proposed algorithm

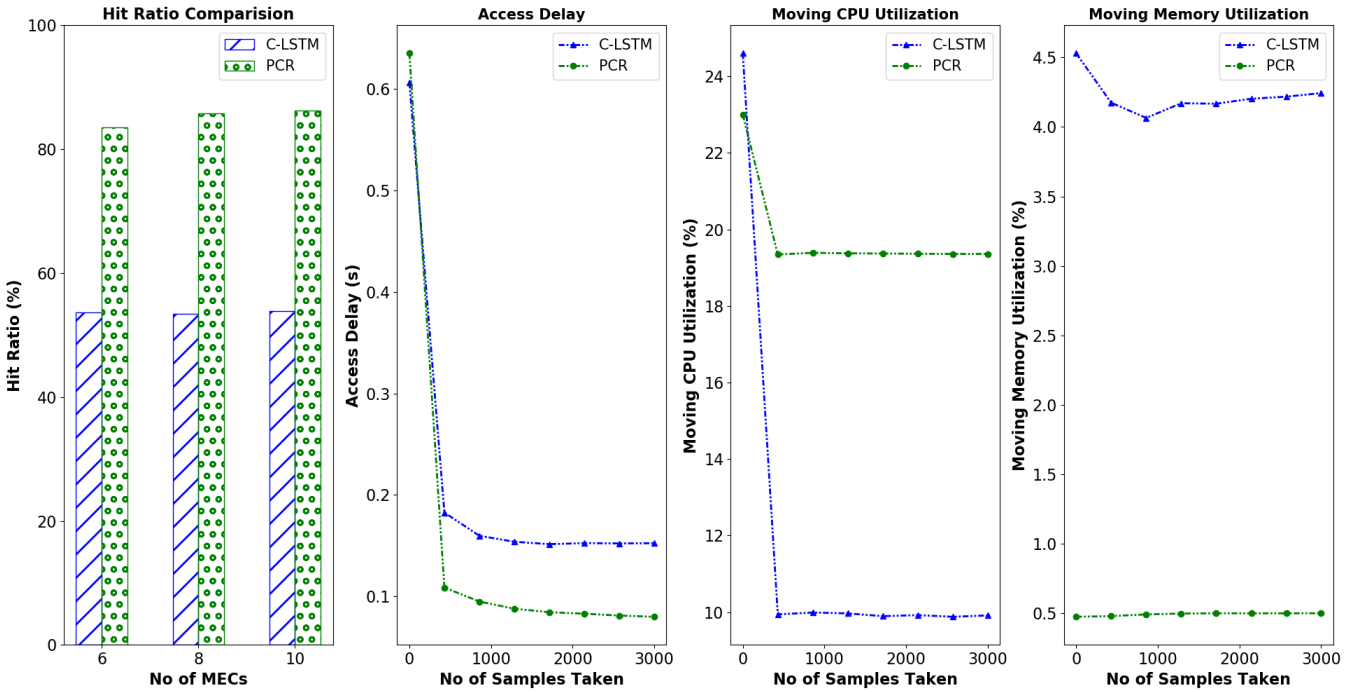

FIGURE 4. Comparison of Hit Ratio, Access Delay, CPU and Memory Utilization of both PCR and C-LSTM

TABLE III. EXPERIMENT SETUP PARAMETERS

General Setup		
Parameters	*Value*	*Meaning*
$\|C_i\|$	$\{6, 8, 10\}$	No of MEC in the collaborative space
$\|R\|$	20,000	Total no of requests
No of content	1070	No of unique content
λ	1	Poisson parameter
Algo 1 parameter		
Parameters	*Value*	*Meaning*
A_{max}	100	Max average frequency
f_{max}	20	frequency count bounding parameter
$cache_{size}$	50	Cache size
h_{size}	$4 \times cache_{size}$	History size
Algo II parameter		
Parameters	*Value*	*Meaning*
w_{size}	$cache_{size} \times .5$	Window size
m	$(w_{size})^2$	Length of history request S
$support_{min}$	0.45	Minimum support
mem_{max}	70%	Maximum memory threshold
$Rule^{max}_{output}$	4	Max length $rule^{ant}$
p	10%	No of top rules selected
Algo III parameter		
Parameters	*Value*	*Meaning*
w	10%	No of MEC that can store a cache

has been compared with a contemporary deep learning-based predictive edge caching algorithm [36]. The authors have used a Recurrent Neural Network model, Long Short-Term Memory (LSTM) to create a model that can make decisions on what to cache on the edge based on cache popularity. Henceforth, this algorithm has been referred to as C-LSTM. C-LSTM has been trained using MovieLens 20M dataset. Therefore, for fairness, the same dataset has been utilized for this experimental comparison. From the MovieLens 20M dataset, the focus was on the movie IDs that have been used in [36] and movie IDs in the range of 1 to 1070. After data preparation and filtering out movie Ids with little references, 444 models have been generated. The generated models have been deployed on the MECs as per the author's specification. The algorithms have been implemented using *Python* and the project is available on Github [39]. In the experimentation, it has been assumed that the arrival time of the user requests on the MEC server follows Poisson distribution $\lambda = 1$. The parameters used for the experiment have been summarised in TABLE III.

A. RESULTS

In this section, the results obtained from the comparison of the two algorithms with respect to hit ratio, access delay, CPU and memory utilization are discussed.

1) Hit Ratio

Fig. 4 shows the hit ratio comparison of PCR and C-LSTM for varying number of MECs. It can be seen that PCR achieves a better hit ratio than C-LSTM in all caches with at least 25% increase. This high increase is due to the effective use of the collaborative cache and its selective caching approach. Additionally, this is also attributed to the efficient identification and replacement of cold cache objects and the ability to learn and predict cache association patterns.

2) Access Delay

The comparison of access delay is depicted in Fig. 4. It can be deduced that PCR obtains lower access delay than C-LSTM.

This is due to the increase in hit ratio as the access delay is dependent on the hit ratio. There is less access delay incurred if the user request is served from the local cache or MEC cache compared to obtaining the request from the content server. Therefore, more hit ratio would lead to lesser access delay. This reduction would lead to better QoE for the end users and a step closer to achieving URLLC.

3) CPU Utilization

The CPU utilization comparison is shown in Fig. 4. C-LSTM uses considerably lower CPU utilization than PCR. This is because C-LSTM is an offline algorithm. Hence, the model has already been trained with the dataset offline and the trained model is then used for caching decisions. Therefore, not much CPU utilization is required for prediction. However, PCR is an online algorithm, therefore, the association prediction is done during runtime and hence obtains a higher CPU utilization. The CPU utilization obtained is stable and predictable. Therefore can be accounted for during live deployment.

4) Memory Utilization

The memory utilization for both C-LSTM and PCR can also be seen in Fig. 4. C-LSTM obtains a higher memory utilization than PCR. However, it is a low percentage compared to the overall memory. The higher memory utilization is because C-LSTM must load each trained model into memory which is then used for prediction. Although PCR will have to store a lot of the parameters in memory, these parameters are capped to prevent overflow and hence achieves lower memory utilization. Low memory utilization is essential in real MEC environments due to the limited computation resources of MEC nodes. The memory utilization obtained during the experiment proofs that the proposed framework can be deployed in real MEC environment.

VI. CONCLUSION

In this paper, a comprehensive study has been done in caching on MEC. Thereafter, PCR scheme has been proposed which is a three-fold caching solution to increase the collaborative hit ratio in the MEC platform and reduce the access delay incurred with obtaining request data. To optimise the hit ratio, access delay and identify cold cache objects, a novel replacement algorithm than can select a victim in constant time has been proposed. Additionally, to dynamically adjust to the ever-changing user request pattern, a proactive predictive caching algorithm to learn cache associations and prefetch cache objects when a user request is anticipated has been presented. Finally, to increase the total hit ratio in the MEC platform, a collaborative caching algorithm for MECs has been described. The proposed PCR scheme has been compared with an existing offline caching algorithm C-LSTM and an extensive experimentation has shown that PCR is better than C-LSTM and other conventional algorithms with regards to hit ratio and reduction of access delay.

VII. FUTURE WORK

The proposed novel scheme PCR, is a proactive distributed caching framework that shares its cache details among collaborative MECs. However, it employs a centralised learning approach where each MEC maintains its predictive model. A complete predictive model of the popular cache objects for each collaborative space can be obtained if a distributed learning scheme is utilized such as Federated Learning. Hence, further research can be done to determine how the proposed predictive algorithm can be decentralised using federated learning.

Additionally, ICN is advocated to shift the communication focus from data location to the data itself by making the named data the priority in the network. Therefore, data can be sourced from the internet using the named data and not the data location or IP address. Exploration analysis can be done to determine if the proposed caching algorithms can be adapted in the context of ICN. In this context, the routers would be used instead of MEC for caching. Therefore, an analysis should be done on where to carry out model training for predictions to optimise latency and avoid overloading of the router's computational resources.

REFERENCES

[1] P. Popovski, J. J. Nielsen, C. Stefanovic, E. De Carvalho, E. Strom, K. F. Trillingsgaard, A.-S. Bana, D. M. Kim, R. Kotaba, J. Park *et al.*, "Wireless access for ultra-reliable low-latency communication: Principles and building blocks," *Ieee Network*, vol. 32, no. 2, pp. 16–23, 2018.

[2] Y. Liu, Z. Zeng, X. Liu, X. Zhu, and M. Z. A. Bhuiyan, "A novel load balancing and low response delay framework for edge-cloud network based on sdn," *IEEE Internet of Things Journal*, 2019.

[3] G. M. D. T. Forecast, "Cisco visual networking index: global mobile data traffic forecast update, 2017–2022," *Update*, vol. 2017, p. 2022, 2019.

[4] T. Taleb, K. Samdanis, B. Mada, H. Flinck, S. Dutta, and D. Sabella, "On multi-access edge computing: A survey of the emerging 5g network edge cloud architecture and orchestration," *IEEE Communications Surveys & Tutorials*, vol. 19, no. 3, pp. 1657–1681, 2017.

[5] A. Mehrabi, M. Siekkinen, and A. Ylä-Jääski, "Qoe-traffic optimization through collaborative edge caching in adaptive mobile video streaming," *IEEE Access*, vol. 6, pp. 52 261–52 276, 2018.

[6] C. Zhang, H. Pang, J. Liu, S. Tang, R. Zhang, D. Wang, and L. Sun, "Toward edge-assisted video content intelligent caching with long short-term memory learning," *IEEE access*, vol. 7, pp. 152 832–152 846, 2019.

[7] A. Silberschatz, P. Galvin, and G. Gagne, *Operating System Concepts, 9th Edition*. Wiley, 2012. [Online]. Available: https://books.google.co.uk/books?id=9VMcAAAAQBAJ

[8] Y. Zhou, J. Philbin, and K. Li, "The multi-queue replacement algorithm for second level buffer caches." in *USENIX Annual Technical Conference, General Track*, 2001, pp. 91–104.

[9] D. Lee, J. Choi, J.-H. Kim, S. H. Noh, S. L. Min, Y. Cho, and C. S. Kim, "Lrfu: A spectrum of policies that subsumes the least recently used and least frequently used policies," *IEEE transactions on Computers*, no. 12, pp. 1352–1361, 2001.

[10] E. J. O'neil, P. E. O'neil, and G. Weikum, "The lru-k page replacement algorithm for database disk buffering," *Acm Sigmod Record*, vol. 22, no. 2, pp. 297–306, 1993.

[11] J. T. Robinson and M. V. Devarakonda, "Data cache management using frequency-based replacement," in *Proceedings of the 1990 ACM SIGMETRICS conference on Measurement and modeling of computer systems*, 1990, pp. 134–142.

[12] T. Johnson, D. Shasha *et al.*, "2q: a low overhead high performance bu er management replacement algorithm," in *Proceedings of the 20th International Conference on Very Large Data Bases*. Citeseer, 1994, pp. 439–450.

[13] J. Liang, D. Zhu, H. Liu, H. Ping, T. Li, H. Zhang, L. Geng, and Y. Liu, "Multi-head attention based popularity prediction caching in social

content-centric networking with mobile edge computing," *IEEE Communications Letters*, pp. 1–1, 2020.

[14] S. Dutta, A. Narang, S. Bhattacherjee, A. S. Das, and D. Krishnaswamy, "Predictive caching framework for mobile wireless networks," in *2015 16th IEEE International Conference on Mobile Data Management*, vol. 1. IEEE, 2015, pp. 179–184.

[15] C. A. Chan, M. Yan, A. F. Gygax, W. Li, L. Li, I. Chih-Lin, J. Yan, and C. Leckie, "Big data driven predictive caching at the wireless edge," in *2019 IEEE International Conference on Communications Workshops (ICC Workshops)*. IEEE, 2019, pp. 1–6.

[16] E. E. Ugwuanyi, S. Ghosh, M. Iqbal, T. Dagiuklas, S. Mumtaz, and A. Al-Dulaimi, "Co-operative and hybrid replacement caching for multi-access mobile edge computing," in *2019 European Conference on Networks and Communications (EuCNC)*. IEEE, 2019, pp. 394–399.

[17] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural Comput.*, vol. 9, no. 8, p. 1735–1780, Nov. 1997. [Online]. Available: https://doi.org/10.1162/neco.1997.9.8.1735

[18] J. N. Van Rijn and F. Hutter, "Hyperparameter importance across datasets," in *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, 2018, pp. 2367–2376.

[19] S. Podlipnig and L. Böszörmenyi, "A survey of web cache replacement strategies," *ACM Computing Surveys (CSUR)*, vol. 35, no. 4, pp. 374–398, 2003.

[20] D. L. Willick, D. L. Eager, and R. B. Bunt, "Disk cache replacement policies for network fileservers," in *[1993] Proceedings. The 13th International Conference on Distributed Computing Systems*. IEEE, 1993, pp. 2–11.

[21] N. Young, "Thek-server dual and loose competitiveness for paging," *Algorithmica*, vol. 11, no. 6, pp. 525–541, 1994.

[22] P. Cao and S. Irani, "Cost-aware www proxy caching algorithms." in *Usenix symposium on internet technologies and systems*, vol. 12, no. 97, 1997, pp. 193–206.

[23] M. Arlitt, L. Cherkasova, J. Dilley, R. Friedrich, and T. Jin, "Evaluating content management techniques for web proxy caches," *ACM SIGMETRICS Performance Evaluation Review*, vol. 27, no. 4, pp. 3–11, 2000.

[24] H. Bahn, S. H. Noh, S. L. Min, and K. Koh, "Using full reference history for efficient document replacement in web caches." in *USENIX Symposium on Internet Technologies and Systems*, 1999.

[25] K. Cheng and Y. Kambayashi, "Lru-sp: a size-adjusted and popularity-aware lru replacement algorithm for web caching," in *Proceedings 24th Annual International Computer Software and Applications Conference. COMPSAC2000*. IEEE, 2000, pp. 48–53.

[26] H. Wu, J. Li, J. Zhi, Y. Ren, and L. Li, "Edge-oriented collaborative caching in information-centric networking," in *2019 IEEE Symposium on Computers and Communications (ISCC)*. IEEE, 2019, pp. 1–6.

[27] A. Ndikumana, S. Ullah, T. LeAnh, N. H. Tran, and C. S. Hong, "Collaborative cache allocation and computation offloading in mobile edge computing," in *2017 19th Asia-Pacific Network Operations and Management Symposium (APNOMS)*. IEEE, 2017, pp. 366–369.

[28] Y. Chen, Y. Liu, J. Zhao, and Q. Zhu, "Mobile edge cache strategy based on neural collaborative filtering," *IEEE Access*, vol. 8, pp. 18 475–18 482, 2020.

[29] X. Zhang and Q. Zhu, "Collaborative hierarchical caching over 5g edge computing mobile wireless networks," in *2018 IEEE International Conference on Communications (ICC)*. IEEE, 2018, pp. 1–6.

[30] J. Liu, D. Li, and Y. Xu, "Collaborative online edge caching with bayesian clustering in wireless networks," *IEEE Internet of Things Journal*, vol. 7, no. 2, pp. 1548–1560, 2019.

[31] Y. M. Saputra, D. T. Hoang, D. N. Nguyen, E. Dutkiewicz, D. Niyato, and D. I. Kim, "Distributed deep learning at the edge: A novel proactive and cooperative caching framework for mobile edge networks," *IEEE Wireless Communications Letters*, vol. 8, no. 4, pp. 1220–1223, 2019.

[32] Y. Chen, Y. Liu, J. Zhao, and Q. Zhu, "Mobile edge cache strategy based on neural collaborative filtering," *IEEE Access*, vol. 8, pp. 18 475–18 482, 2020.

[33] J. Xu, K. Ota, and M. Dong, "Energy efficient hybrid edge caching scheme for tactile internet in 5g," *IEEE Transactions on Green Communications and Networking*, vol. 3, no. 2, pp. 483–493, 2019.

[34] K. Qi and C. Yang, "Popularity prediction with federated learning for proactive caching at wireless edge," in *2020 IEEE Wireless Communications and Networking Conference (WCNC)*. IEEE, 2020, pp. 1–6.

[35] Y. Qian, R. Wang, J. Wu, B. Tan, and H. Ren, "Reinforcement learning-based optimal computing and caching in mobile edge network," *IEEE Journal on Selected Areas in Communications*, vol. 38, no. 10, pp. 2343–2355, 2020.

[36] S. Rahman, M. G. R. Alam, and M. M. Rahman, "Deep learning-based predictive caching in the edge of a network," in *2020 International Conference on Information Networking (ICOIN)*. IEEE, 2020, pp. 797–801.

[37] "MovieLens 20M Dataset," Sep. 2015. [Online]. Available: https://grouplens.org/datasets/movielens/20m/

[38] H.-G. Song, S. H. Chae, W.-Y. Shin, and S.-W. Jeon, "Predictive caching via learning temporal distribution of content requests," *IEEE Communications Letters*, vol. 23, no. 12, pp. 2335–2339, 2019.

[39] E. E. Ugwuanyi, "emylincon/caching," Jul. 2020, original-date: 2020-06-17T14:13:19Z. [Online]. Available: https://github.com/emylincon/caching

[40] U. Emeka, "CacheTest." [Online]. Available: https://cachetrace.herokuapp.com/

[41] N. Zhou, M. Qiao, and J. Zhou, "Bi_apriori algorithm: research and application based on battery production data," in *2019 IEEE 9th International Conference on Electronics Information and Emergency Communication (ICEIEC)*. IEEE, 2019, pp. 1–5.

[42] R. Agrawal and R. Srikant, "Fast algorithms for mining association rules in large databases, vldb'94: Proceedings of the 20th international conference on very large data bases," *San Francisco, CA, USA*, pp. 487–499, 1994.

[43] J. Han, J. Pei, Y. Yin, and R. Mao, "Mining frequent patterns without candidate generation: A frequent-pattern tree approach," *Data mining and knowledge discovery*, vol. 8, no. 1, pp. 53–87, 2004.

[44] B. Wu, D. Zhang, Q. Lan, and J. Zheng, "An efficient frequent patterns mining algorithm based on apriori algorithm and the fp-tree structure," in *2008 Third International Conference on Convergence and Hybrid Information Technology*, vol. 1. IEEE, 2008, pp. 1099–1102.

[45] "Netlify: All-in-one platform for automating modern web projects." [Online]. Available: https://www.netlify.com/